Assignment 2 (Algorithm notes)

1 Overview

Let’s quickly overview what you must do for this assignment:

1. You must construct a cube, cylinder, cone, and sphere using just triangles, correctly handling the tessellation parameters for the shape.

2. You must correctly handle the normals for the different shapes. Shapes with smooth surfaces (i.e., cylinder and sphere) must use “smooth” triangles (and thus will have have three normals per triangle).

3. You must not re-tessellate the shape if it has not changed; just redraw the triangles.

I hope the rest of the document can provide additional help to your understanding of the project.

2 Tessellation

2.1 Cube

Parameter 1: the number of subdivisions along an edge of the cube

Tessellating a cube is simple. The best way to start out tessellating a cube is to assume the first parameter is set to 1. This will result in having just two triangles per face. Just hard code these 12 triangles (6 faces \( \times \) 2 triangles = 12 triangles). Once you have that working, modify your tessellation routine so that it will tile each face with smaller versions of the simple 2-triangle face.

2.2 Cylinder

Parameter 1: slices (the number of radial divisions)
Parameter 2: stacks (the number of divisions height-wise)

Tessellating a cylinder uses what we learned in class for parameterizing shapes. For example, the top and bottom caps are tessellated using circle parameterization, and the side is tessellated using cylindrical surface parameterization. Your tessellation should break the caps into a fan of triangles (determined by slices), and break the barrel into a grid of triangles (whose rows and columns are determined by stacks and slices).

A very important thing to realize (which will cause you hours of agony and pain when debugging if you don’t), is that the \( \sin() \) and \( \cos() \) (and while we’re at it, \( \tan() \)) functions take their arguments in radians, not degrees. A degree is equal to \( \pi/180 \) radians. So \( \theta \) should run from 0 to \( 2\pi \) radians which is 0 to 360 degrees.
2.3 Cone

Parameter 1: slices (the number of radial divisions)
Parameter 2: stacks (the number of divisions height-wise)

Tessellating a cone is almost exactly like tessellating a cylinder, the only difference is that the top cap of the cylinder is shrunk to a point in the cone. Likewise, the top row of triangles that would have been on the cylinder barrel would have to be re-designed in the cone into a fan of triangles (determined by slices).

2.4 Sphere

Parameter 1: slices (the number of radial divisions)
Parameter 2: stacks (the number of divisions height-wise)

Sphere tessellation is probably the hardest part of this assignment. There are also radically different ways of doing it. The simpler way is what we learned in class: longitude (slices) and latitude (stacks). Be careful about the top and bottom row of triangles, they should be fans that surround the two poles.

One thing you may notice about this way of tessellating a sphere is you get drastically different sized triangles as you move across its surface, even though the sphere has complete radial symmetry. A better, consistent tessellation can be achieved by noticing that an icosahedron looks strikingly similar to a sphere. In fact, you can subdivide each of the faces of an icosahedron, and it begins to look more and more like a sphere (check out the demo)! Exploring this idea may lead you to having a really cool sphere tessellator, which will earn you extra credit. (Hint: this method will only use one of the tessellation parameters.)

3 Speed

As was mentioned a couple of times already, recalculating all the triangles (and normals) needed to construct a shape every time it is redrawn will cause your program to be painfully slow. What you will need to do is whenever the user modifies some parameter that affects how the shape is tessellated, re-tessellate the shape when it is drawn. All other changes should simply “replay” the tessellation from a list that you create when you really tessellated the shape. Such a list should probably contain the triangles used to create the shape. For each triangle, store the vertices, the normal(s), as well as some indicator as to the type of triangle it is. This list can be implemented in many ways— how it is done is left up to you. The most efficient method probably is simply to allocate a large block of memory, and just treat it as one enormous array.

3.1 Object hierarchy

It will be very useful to have a base class which has the following operations:

- Flag if triangles have a normal per vertex or one normal for the whole triangle.
- AddTriangleWithVertexNormals
- AddTriangleWithFaceNormal
- DrawTriangles
- NewTessellation(int, int)
- (optional) SetNumberTriangles(int)
- (optional) AddSquare(); splits triangle for you
- (optional) AddTriangle(); calculates face normal using cross product
• (optional) SplitAllTriangles()

This class is responsible for storing the triangles and normals. Sub-classes, e.g., the sphere, cube, and cylinder, are then responsible just for generating the triangles.